Assignment 3 writeup

Comparisons of Different LZW Implementation’s Compressed files

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Text File | Original  File | LZW size | LZW ratio | LZWmod without reset | LZWmod without reset ratio | LZWmod with reset | LZWmod with reset ratio | Unix size | Unix ratio |
| all.tar | 3,031,040 B | 1,846,854 B | 1.64 | 1,792,781 B | 1.69 | 1,792,781 B | 1.69 | 1,179,467 B | 2.56 |
| assig2.doc | 87,040 B | 74,574 | 1.17 | 40,039 B | 2.17 | 40,039 B | 2.17 | 40,040 B | 2.17 |
| bmps.tar | 1,105,920 B | 925,079 B | 1.19 | 80,913 B | 13.67 | 80,913 B | 13.67 | 80,913 B | 13.67 |
| code.txt | 72,351 B | 30,980 B | 2.34 | 24,544 B | 2.95 | 24,544 B | 2.95 | 24,545 B | 2.95 |
| code2.txt | 57,701 B | 24,138 B | 2.39 | 20,515 B | 2.81 | 20,515 B | 2.81 | 20,516 B | 2.81 |
| edit.exe | 236,328 B | 250,742 B | 0.94 | 156,409 B | 1.51 | 156490 B | 1.51 | 151,111 B | 1.56 |
| frosty.jpg | 126,748 B | 177,453 B | 0.71 | 163,789 B | 0.77 | 163,789 B | 0.77 | 126,750 B | 1.00 |
| gone\_fishing.bmp | 17,336 B | 9,278 B | 1.87 | 8,962 B | 1.93 | 8,962 B | 1.93 | 8,962 B | 1.93 |
| large.txt | 1,220,703 B | 605,184 B | 2.02 | 501,777 B | 2.43 | 501,777 B | 2.43 | 522,673 B | 2.34 |
| Lego-big.gif | 93,371 B | 128,973 B | 0.72 | 122,493 B | 0.76 | 122,493 B | 0.76 | 93,371 B | 1.00 |
| medium.txt | 25,407 B | 13,197 B | 1.93 | 12,530 B | 2.02 | 12,530 B | 2.02 | 12,531 B | 2.03 |
| texts.tar | 1,382,400 B | 101,2179 B | 1.37 | 597,847 B | 2.31 | 597,847 B | 2.31 | 589,697 B | 2.34 |
| wacky.bmp | 921,654 B | 4,302 B | 214.24 | 3,951 B | 233.27 | 3,951 B | 233.27 | 3,952 B | 233.21 |
| winnt256.bmp | 157,044 B | 159,050 B | 0.99 | 62,931 B | 2.50 | 62,931 B | 2.50 | 62,931 B | 2.50 |

For this assignment, the purpose was to understand the LZW compression algorithm, its performance and its implementation. I modified the author’s, Robert Sedgewick, implementation of the LZW compression algorithm. There were four goals for this assignment that I was tasked to complete. The first goal was to read the input file as a stream of bytes instead of all at once, and the second was to avoid the theta(n) overhead of using String.substring(). The last two goals were to allow the codebook size to increase beyond the 4096 entries in the textbook’s implementation using adaptive codeword width, and to allow LZW to learn new patterns after the codebook size is reached by giving the user the option to reset the codebook. In the TSTmod.java file, originally the author was using Strings, and he used 12 as the codeword’s width, and 2^12 as the number of codewords in the codebook. In my modified version of the LZW algorithm, I use StringBuilders, 16 as the codeword’s width, and 2^16 as the number of codewords in the codebook. The option to reset was also included when all of the 65536 codewords were used. Also, the modified version uses adaptive codeword width so it made a bigger symbol table, while the original did not.

When comparing all four of the LZW variation programs to each other by looking at the compression ratio of different files, it is shown that in most cases, the predefined unix compress program has larger compression ratios than the ratios of other implementations. There were some instances where the compression ratios matched with the LZW with and without reset, but overall, the unix compression program had the best compression. For the unix compression program, the best compression ratio was of wacky.bmp. It has a ratio of 233.21, which is pretty high. This is a good thing since the file did not have that much in it. The worst ratios are of Lego-big.gif and frosty.jpg, which both have ratios of 1.00. This is because since compress checks if the compression ration will improve when it runs, if it does not improve, it does not compress. For the unix program, the sizes of Lego-big.gif and frosty do not change from the original file sizes. The compressed size equals the original size for these two files. The higher the compression ratios the more efficient, which means that the program will compress with a speedier performance and compilation.

The second ranked best implementation is tied. My LZWmod with reset and LZWmod without reset had all the same file sizes and compression ratios. There may have been an error in my code regarding this a couple files because the all.tar’s file size for LZWmod with reset is much larger than what it should be, based off the autograder. Resetting the code is supposed to have more of an effect on the sizes and ratios. I speculate that in my code, I may have missed checking for an instance where the codebook has to be reset, but I was not able to figure out exactly what the issue was for that all.tar file. But for most of the other files, it is correct that with reset and without reset did not make any difference in the compression ratios. Between the LZW implementation with reset and without reset, the best compression ratios were of wacky.bmp, which had a ration of 233.27. The file did not have much in it, so this was not surprising. The worst for both with and without reset were again Lego-big.gif and frosty.jpg, since they are images that have a lot of details in them.

Lastly, the author’s original LZW implementation had the worst compression ratios out of the 4 total implementations. I believe that it would not be very efficient for files that have a large amount of content because it reads in each string, which would take a long time. Also, another issue is that it uses substring, and that makes it time consuming, which is why it was a goal that needed to be completed in the modified version. The worst compression ratio of the author’s LZW implementation was Lego-big.gif and frosty.jpg, and the best was wacky.bmp. The overall lowest and worst ratio was of the author’s implementation, which was of frosty.jpg (0.71). I believe this was the worst ranked out of the total 4 because it used the codeword’s length as 12. It does not use adaptive codeword width, so it is possible it ran out of codewords.